Acta Polytechnica Vol. 48 No. 4/2008

Acceleration of Sparse Matrix-Vector Multiplication by Region Traversal

I. Šimeček

Sparse matrix-vector multiplication (shortly SpM×V) is one of most common subroutines in numerical linear algebra. The problem is that the memory access patterns during SpM×V are irregular, and utilization of the cache can suffer from low spatial or temporal locality. Approaches to improve the performance of SpM×V are based on matrix reordering and register blocking. These matrix transformations are designed to handle randomly occurring dense blocks in a sparse matrix. The efficiency of these transformations depends strongly on the presence of suitable blocks. The overhead of reorganization of a matrix from one format to another is often of the order of tens of executions of SpM×V. For this reason, such a reorganization pays off only if the same matrix A is multiplied by multiple different vectors, e.g., in iterative linear solvers.

This paper introduces an unusual approach to accelerate SpM×V. This approach can be combined with other acceleration approaches and consists of three steps:

1) dividing matrix A into non-empty regions,

2) choosing an efficient way to traverse these regions (in other words, choosing an efficient ordering of partial multiplications),

3) choosing the optimal type of storage for each region.

All these three steps are tightly coupled. The first step divides the whole matrix into smaller parts (regions) that can fit in the cache. The second step improves the locality during multiplication due to better utilization of distant references. The last step maximizes the machine computation performance of the partial multiplication for each region.

In this paper, we describe aspects of these 3 steps in more detail (including fast and time-inexpensive algorithms for all steps). Our measurements prove that our approach gives a significant speedup for almost all matrices arising from various technical areas.

Keywords: cache hierarchy, sparse matrix-vector multiplication, region traversal.

Introduction

There are several formats for storing sparse matrices. They have been designed mainly for *SpM×V*. *SpM×V* for the most common format, the *compressed sparse rows* (shortly CSR) format (see Section 1.2.2), suffers from low performance due to indirect addressing. Several studies have been published about increasing the efficiency of *SpM×V* [1, 2, 3].

There are some formats, such as *register blocking* [4], that eliminate indirect addressing during *SpM×V*. Then, vector instructions can be used. These formats are suitable only for matrices with a known structure of nonzero elements.

The processor cache memory parameters must also be taken into account. The overhead of matrix reorganization from one format to another is often of the order of tens of executions of *SpM×V*. Such a reorganization pays off only if the same matrix A is multiplied by multiple different vectors, e.g., in iterative linear solvers.

Many authors (such as [5]) have overlooked the overhead of matrix transformation and have designed time-expensive matrix storage optimization. In contrast to them, we try to find a near-optimal matrix storage format to maximize the performance of *SpM×V* with respect to matrix transformation overhead and cache parameters.

1 Terminology and notation

1.1 The cache model

The cache model that we consider here corresponds to the structure of L1 and L2 caches in the Intel x86 architecture. An *s-way set-associative* cache consists of h *sets*, and one set consists of s independent *blocks* (called *lines* in Intel terminology). Let C_S denote the size of the data part of a cache in bytes, and let B_S denote the cache block size in bytes. Then $C_S = s \cdot B_S \cdot h$. Let S_D denote the size of type `double` and S_I the size of type `integer` in bytes.

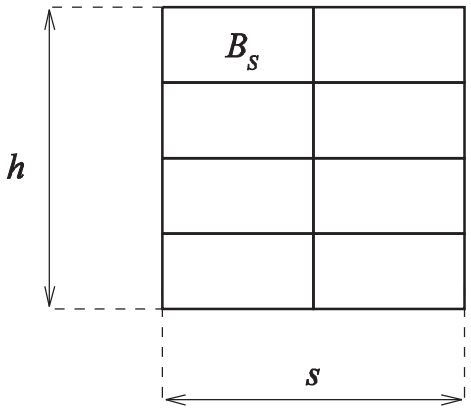

Fig. 1: Description of cache parameters

We consider two types of cache misses: (1) *Compulsory* misses (sometimes called *intrinsic* or *cold*), which occur if the required memory block is not in the cache because it is being accessed for the first time, and (2) *thrashing* misses (also called *cross-interference*, conflict, or *capacity* misses), which occur if the required memory block is not in the cache even though it was previously loaded, because it has been replaced prematurely from the cache for capacity reasons.

1.2 Common sparse matrix formats

In the following text, we assume that A is a real sparse matrix of order n and x, y are vectors of size n. We consider the

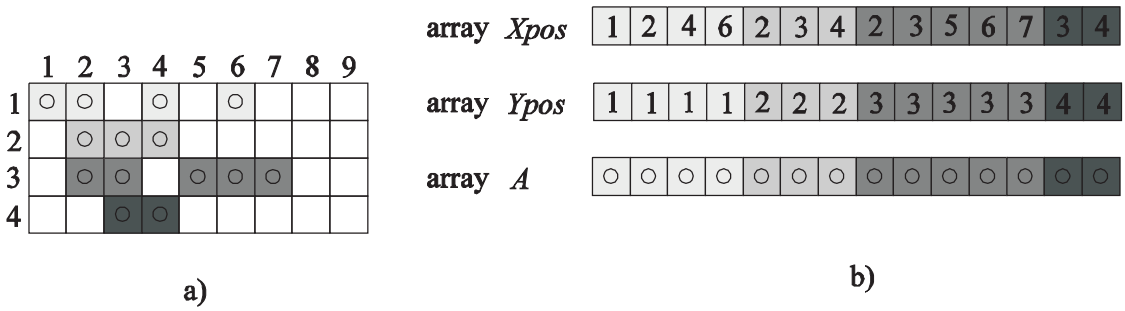

Fig. 2: An example of a sparse matrix A in a) dense format, b) coordinate (XY) format

operation of $SpM{\times}V$ $Ax = y$. Let n_Z denote the total number of nonzero elements in A.

1.2.1 Coordinate (XY) format

In this simplest sparse format, a matrix A is represented by 3 linear arrays ***A***, ***Xpos***, and ***Ypos***. Array ***A*** stores the nonzero elements of A, and arrays ***Xpos*** and ***Ypos*** contain x-positions and y-positions, respectively, of the nonzero elements.

The storage complexity of the XY format is $n_Z(S_D + 2S_I)$, so the XY format can be space inefficient.

Locality during *SpM×V* in the XY format

During one $SpM{\times}V$ in the XY format, every element of

- arrays ***A***, ***Xpos***, ***Ypos*** is read only once;
- output array ***y*** is *written repeatedly*;
- input array ***x*** is *read repeatedly*.

In comparison to the CSR format (see Section 1.2.2), this causes too many compulsory misses for regular matrices.

1.2.2 Compressed sparse row (CSR) format

This is the most common format for storing sparse matrices. A matrix A stored in the CSR format is represented by 3 linear arrays ***A***, ***adr***, and ***ci***. Array ***A*** stores the nonzero elements of A, array ***adr***$[1, \ldots, n]$ contains indices of initial nonzero elements of the rows of A, and array ***ci*** contains column indices of nonzero elements of A. Hence, the first nonzero element of row j is stored at index ***adr***$[j]$ in array A. Its storage complexity is $n_Z(S_D + S_I) + nS_I$. Every regular matrix must contain at least one element per every row. Hence, $n \leq n_Z$. From the storage complexities of the two formats, we can conclude that for every regular matrix the CSR format is the same or more space-efficient than the XY format.

Locality during *SpM×V* in CSR format

During one $SpM{\times}V$ in CSR format, every element of

- array ***A***, ***ci***, ***adr*** is read only once;
- output array ***y*** is written *only once*;
- input array ***x*** is *read repeatedly*.

Some reads (writes) can conflict with other reads (writes), and thrashing misses may occur. The CSR format can be inefficient due to many *thrashing misses* for some types of matrices:

- matrices with a large number of nonzero elements per row (relative to the cache size),
- matrices with large bandwidth (relative to the cache size).

2 Increasing locality in *SpM×V*

2.1 Usual approach

Sparse matrices often contain dense submatrices (called blocks), so various blocking formats have been designed to accelerate matrix operations (mainly $SpM{\times}V$).

Compared to CSR format, the aim of these formats is to consume less memory and to allow better use of registers. The effect of indirect addressing is also to be reduced, and vector (SIMD) instructions can be used.

2.2 Our approach

In general, there is only one way to improve the locality in $SpM{\times}V$: by making the multiplication more cache-sensitive.

In a typical numerical algorithm, the cache hit ratio is high if the cache can hold data accessed in the innermost loop of the algorithms. Memory accesses that do not satisfy this assumption are called *distant references*. In this paper, we propose

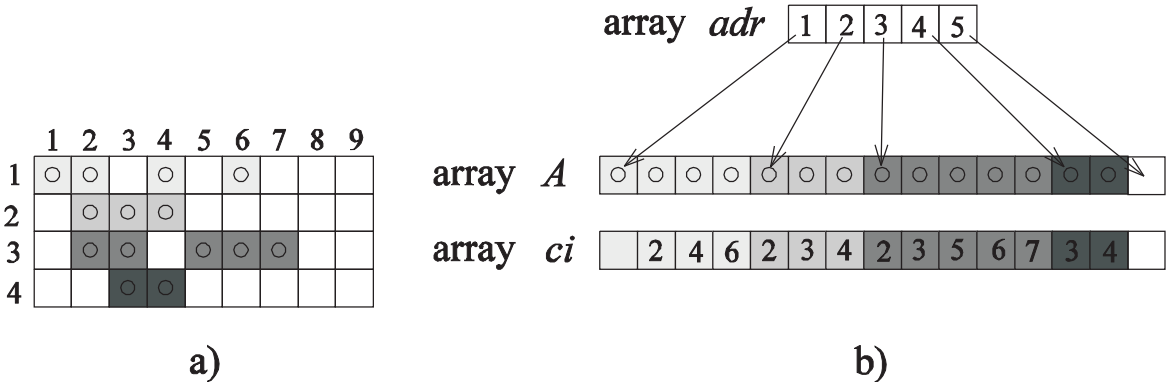

Fig. 3: An example of a sparse matrix A in a) dense format, b) compressed sparse row format

and explore a new method for increasing the locality of *SpM×V*, consisting of 3 steps:

1) Divide matrix A into disjoint nonempty rectangular submatrices (*regions*).
2) Choose a suitable storage format for the regions.
3) Choose a good traversal of these regions during *SpM×V*.

2.2.1 Dividing the matrix into regions

We divide matrix A into disjoint nonempty rectangular *regions* so that the cache can hold all data accessed during partial *SpM×V* within each region. The whole matrix A is represented by two data structures:

- R-aux = auxiliary data of regions.
  - R-x, R-y = position of the beginning of the region,
  - R-s = number of nonzero elements inside the region,
  - R-addr = pointer into R-data.
- R-data = data inside regions. It contains the values and locations of nonzero elements.

In this paper, we assume for simplicity that regions are square submatrices of the same size r containing at least one nonzero element. An optimal value of r is hard to predict, because it depends on the locations of nonzero elements and cache sizes. The value of the region side size r must be chosen to address the following trade-off:

- greater regions lead to a smaller region storage overhead,
- smaller regions lead to higher cache locality, but the computational complexity of an optimal traversal algorithm (see later) grows dramatically with the number of regions.

In this paper, we leave this problem open and choose $r = 240$, so that the values of the coordinates of the elements in the regions can be represented in one byte.

2.2.2 Choosing a suitable storage format for the regions

The second important decision is to choose an efficient format for the R-data structure.

Regions are viewed as submatrices. Therefore, we can use common sparse matrix formats for storing nonzero elements inside them. We can use the information that an element is inside a region, so we can express all coordinates **relative** to the region. All coordinates can be stored by using smaller data-types (*char* or *byte*) instead of *long int*. This leads to **compressed representation** of the input matrix. We call these new formats *R-based*. In the following text, we will consider only the R-CSR and R-XY formats:

- R-XY, derived from the XY storage format.
- R-CSR, derived from the CSR storage format.

Each region is stored either in R-XY or R-CSR format to minimize compulsory misses (see Section 1.2.2). Let n' be the number of nonzero elements within a region. If $n' \geq r$, then the R-CSR format is chosen, otherwise the R-XY format is chosen. One possible drawback of these formats is that in some architectures (including Intel x86 and compatible CPUs), data memory transfers with smaller data-types are very slow.

2.2.3 Choosing a good traversal of regions

For the performance of a program, a high cache hit ratio (high locality) is crucial. Better locality in *SpM×V* is only one step toward higher performance of *SpM×V*. Choosing a good traversal is equivalent to finding a good ordering of regions (partial multiplications). We can simply say that the traversing is good if it leads to a fast *SpM×V*. Now we will discuss this obvious assumption in a more formal way with an in-depth analysis.

The minimizing of the number of cache misses (denoted by L) is equal to the problem of optimal indexing of regions (denoted by V). It is an NP-complete problem, so finding an optimal solution is hard. We try to keep our approach efficient, so we have considered only a number of standard region traversals:

- row-wise traversing (see Fig. 4a)),
- snake-wise row traversing (see Fig. 4b)),
- column-wise traversing (see Fig. 4c)),
- snake-wise column traversing (see Fig. 4d)),
- diagonal traversing (see Fig. 4e)),
- reversal diagonal traversing (see Fig. 4f)),

and two recursive region traversals

- Z-Morton traversing (see Fig. 5a)),
- C-Morton traversing (see Fig. 5b)).

A very important question is how to measure the performance of SpM×V, and the effect of various traversal strategies. We have used 3 approaches:

1. Real execution time measurements.
   - The results are platform dependent.
   - They can be influenced by other OS processes.

   All HW and SW aspects are taken into account.
2. Estimating of the number of cache misses, using a SW cache analyzer (for example [6], which uses a cache model, as described in Section 1.1).
   - It cannot be influenced by other OS processes.
   - This solution takes only the cache memory effect into account.
3. Estimations of the number of cache misses by simulation algorithms (see Section 2.3.1).
   - This is a fast way.
   - The simulation algorithms are slightly inexact, due to their initial assumptions.
   - This solution takes only the cache memory effect into account.

A cache behavior simulation algorithm

We have designed a cache behavior simulation algorithm that predicts the number of cache misses during an execution of *SpM×V*. The number of cache misses depends on the cache-block replacement strategy; we assume the LRU replacement strategy as the most common case. For different schemas of the cache-block replacement strategy, similar simulation algorithms can be derived.

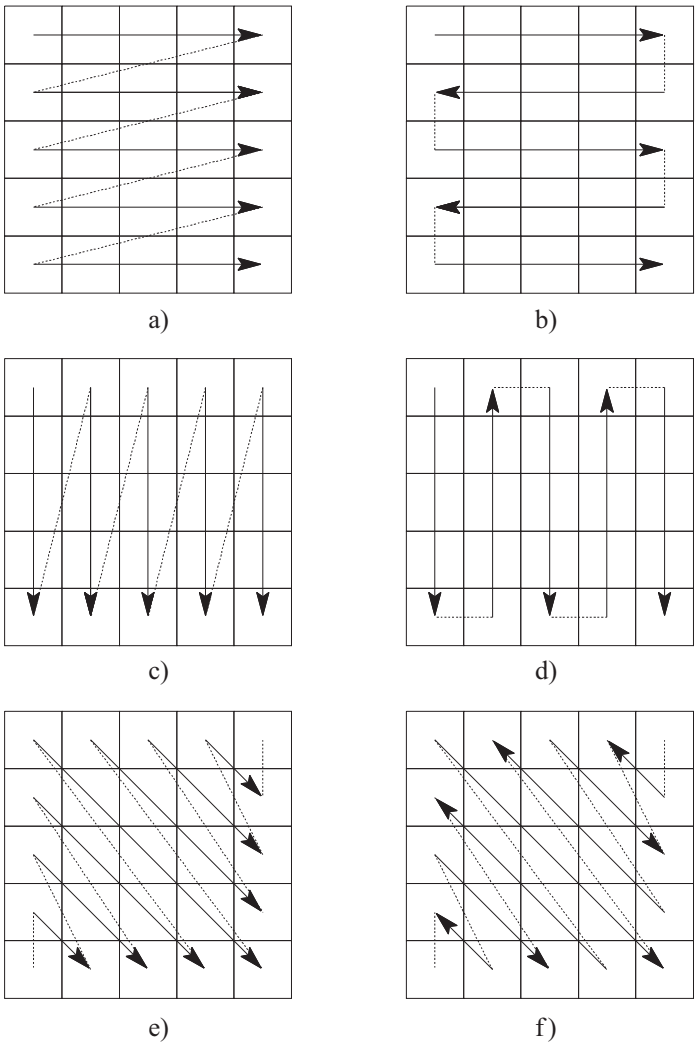

Fig. 4: The most "typical" region traversals

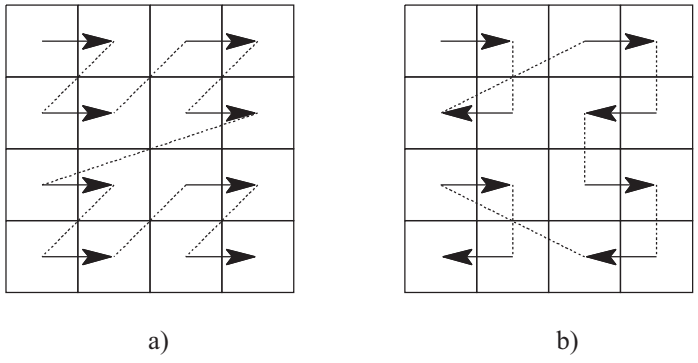

Fig. 5: Recursive region traversals

A cache behavior simulation algorithm (CBSA) is an even more abstract model than CS [6]. It uses the queue Q for modeling a cache memory. For a chosen traversal ordering π and a chosen cache line replacement strategy, this allows us to estimate the number of cache misses during the execution of $SpM{\times}V$. CBSA assumes a fully associative cache memory, omits cache spatial locality, and counts only load operations with parts of array $\boldsymbol{x}$ and $\boldsymbol{y}$, because only these arrays are reused.

Definition of region traversing

- Let $G=(V,E)$, $|V|=m$, be a subgraph of a 2D mesh $(n{\times}n)$ such that every vertex $V_i \in V$ has coordinates $X(V_i)$, $Y(V_i)$ and the value $l(V_i)$. (Each vertex represents a region of the matrix containing at least one nonzero element; the value $l(V_i)$ represents the number of nonzero elements inside this region).

- π is a permutation of $[1 \dots m]$ defining a traversal of G so that the ith traversed block is $V_{\pi(i)}$.
- Let $L(\pi)$ be the number of cache misses if G is traversed according to π.
- The optimal traversal is a permutation π with minimal $L(\pi)$.

The algorithm

CBSA for the LRU replacement strategy

(1) $L = 0$;
(2) **for** $i = 1$ **to** m
(3) $U = V_{\pi(i)}$
(4) $X_{\text{temp}} = [\text{'}x\text{'}, X(U)]$;
(5) $Y_{\text{temp}} = [\text{'}y\text{'}, Y(U)]$;
(6) **if** $X_{\text{temp}} \notin Q$ **then** $L = L + l(U)$;
(7) **if** $X_{\text{temp}} \notin Q$ **else** remove X_{temp} from Q;
(8) store X_{temp} into Q;
(9) **if** $Y_{\text{temp}} \notin Q$ **then** $L = L + \min(r, l(U))$;
(10) **if** $Y_{\text{temp}} \notin Q$ **else** remove Y_{temp} from Q;
(11) store Y_{temp} into Q;
(12) **while** size of $Q > C_S$ **do** dequeue(Q);

;;The result is stored in the variable L.

Comments on the algorithm

- Two operations with the queue appear:
  - remove X from Q: this operation removes element X from the queue Q irrespective of its position.
  - dequeue (Q): this operation removes the element at the front of the queue Q.
- In codelines (4) and (5), special additional flags 'x' and 'y' serve for distinguishing memory operations with arrays $\boldsymbol{x}$ and $\boldsymbol{y}$.
- In codeline (9), cache misses are simulated. A part of the array $\boldsymbol{x}$ or $\boldsymbol{y}$ must be loaded, because they are new or they were replaced prematurely from the cache for capacity reasons. The number of cache misses is equal to r (for the R-CSR format) or $l(U)$ (for the R-XY format).

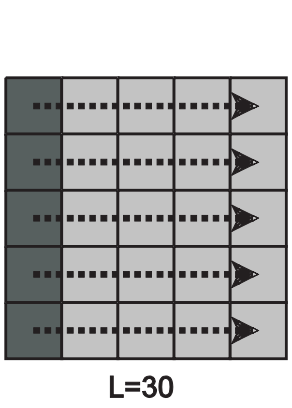

Fig. 6: Example of the CBS algorithm. A dark box means 2 misses, a light box means 1 miss, a white box means no miss. Traversing the matrix in row-wise traversing causes 30 misses.

- In codelines (7) and (10), refreshments of the cachelines are described.
- In codeline (12), the queue size is decreased to the cache size. Elements are freed in the order according to the LRU strategy.

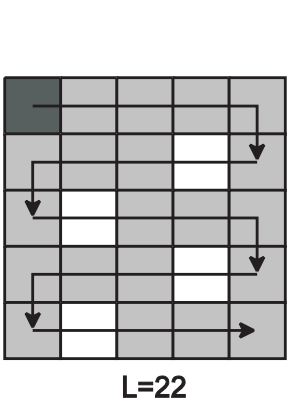

Fig. 7: Example of the CBS algorithm. A dark box means 2 misses, a light box means 1 miss, a white box means no miss. Traversing the matrix in snake-like row-wise traversing

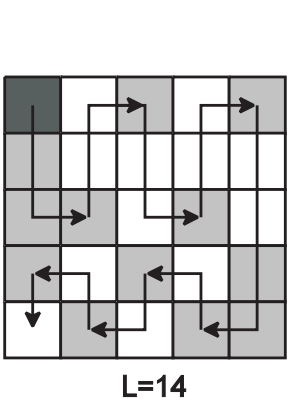

Fig. 8: Example of the CBS algorithm. A dark box means 2 misses, a light box means 1 miss, a white box means no miss. Traversing the matrix in area-filling traversing causes 14 misses.

3 Evaluation of the results

All results were measured on Pentium Celeron 2.4 GHz, 512 MB@ 266 MHz, running OS Windows XP with the following cache parameters:

The L1 cache is a data cache with $B_S = 64$, $C_S = 8K$, $s = 4$, $h = 32$, and LRU replacement strategy.

The L2 cache is unified with $B_S = 64$, $C_S = 128K$, $s = 4$, $h = 1024$, and LRU strategy.

Also, $S_D = 8B$, $S_I = 4B$,

SW: Microsoft Visual C++ 6.0 Enterprise edition

Intel compiler version 7.1 with switches: `-O3 -Ow -Qpc64 -G6 -QxK -Qipo -Qsfalign16 -Zp16`

All cache events were monitored by the Intel Vtune performance analyzer 7.0 and verified by the Cache Analyzer [6].

3.1 Test data

We have 3 sources of test data

- We wrote a GEN program that generates symmetric, positive definite matrices produced by discretization of two elliptic partial differential equations with the Dirichlet boundary condition on rectangular grids. For testing purposes, we used 4 matrices
  1. Matrix A is a random banded sparse matrix with $n = 2 \cdot 10^4$; $n_Z = 8.7 \cdot 10^4$.
  2. Matrix B is filled by randomly located short ($l < 10$) diagonal blocks with $n = 2 \cdot 10^4$; $n_Z = 3 \cdot 10^5$.
  3. Matrix C is given by discretization on a 100×100 rectangular grid with $n = 2 \cdot 10^4$; $n_Z = 2.6 \cdot 10^5$.
  4. Matrix D is given by discretization on a 10×1000 rectangular grid with $n = 2 \cdot 10^4$; $n_Z = 2.4 \cdot 10^5$.
- We used 41 real matrices from various technical areas from MatrixMarket and the Harwell sparse matrix test collection.
- We wrote a SYN program that generates random sparse matrices with the following parameters: $1000 \le n \le 10000$, $1000 \le n_Z \le 500000$.

3.2 Performance metrics

3.2.1 Compression rate

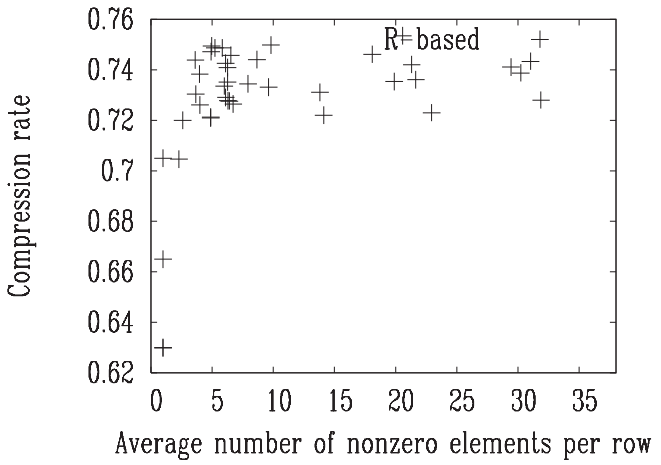

Fig. 9: The compression rate

We define the parameter *Compression rate* as the ratio between the space complexity of the R-based format and the space complexity of the CSR format.

The decision whether a given region should be stored in the R-XY format or in the R-CSR format can be made statically (see Section 2.2.2). From Fig. 9, we can conclude that the average compression rate is about **74 %**. The only exception is matrices with the average number of nonzero elements per row close to 1. For such matrices, the compression rate varies from 63 % (for small matrices) to 74 % (large matrices). This compression rate variability is caused by the array ***adr*** overhead in the CSR format.

3.2.2 Accuracy of CBSA

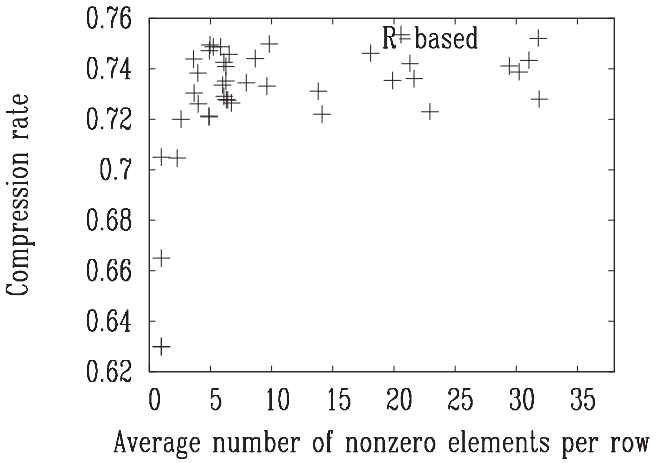

Fig. 10: Accuracy of CBSA

We define the accuracy of CBSA as the ratio between the number of cache misses predicted by CBSA and the number of cache misses measured by the SW cache analyzer.

Fig. 10 illustrates that the average accuracy of CBSA is 93 % for the L1 cache and 95 % for the L2 cache.

3.2.3 Performance

We define *Performance* as the ratio between the number of FPU operations during *SpM×V* and the execution time of *SpM×V*.

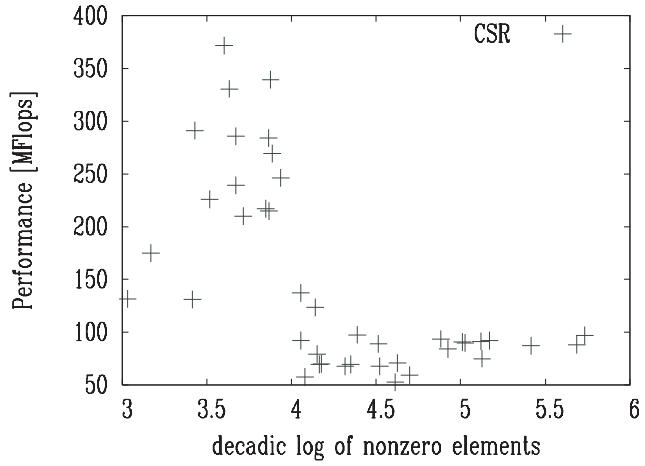

Fig. 11: Performance of *SpM×V* with the CSR format

If the cache is flushed out before the measurement and the multiplication is done only once, the CSR format and the R-based format achieve almost the same performance. This is due to the fact that during one multiplication, the impact of cache re-use is negligible.

Figs. 11 and 12 illustrate the performance of *SpM×V* with the CSR format and the R-based format, respectively. In this case, multiplication is done repeatedly (1000 times). There is a performance gap between small matrices that fit in the cache and large matrices that do not fit in the cache.

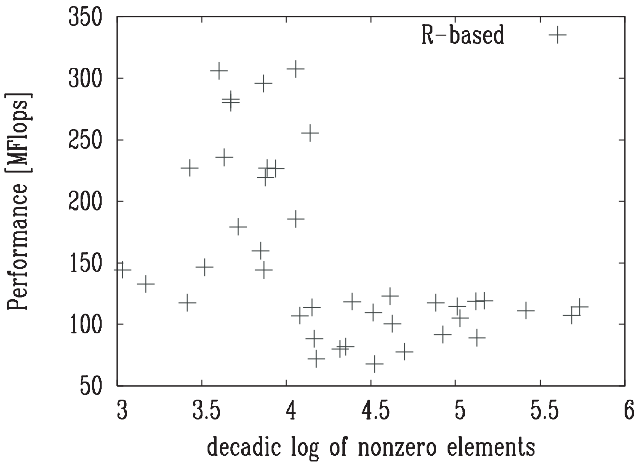

Fig. 12: Performance of $SpM{\times}V$ with the R-based format

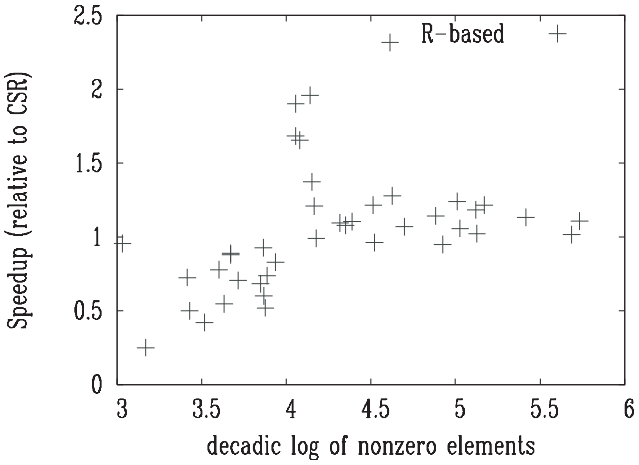

Fig. 15: The speedup estimated by a CBSA

3.2.4 Speedup

We define *Speedup* as the ratio between the time of $SpM{\times}V$ with the CSR format and the time of $SpM{\times}V$ with the R-based format.

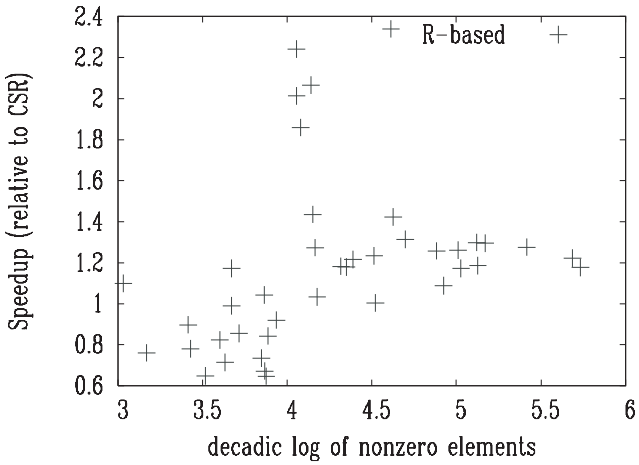

Fig. 13: The speedup measured by real execution time measurements

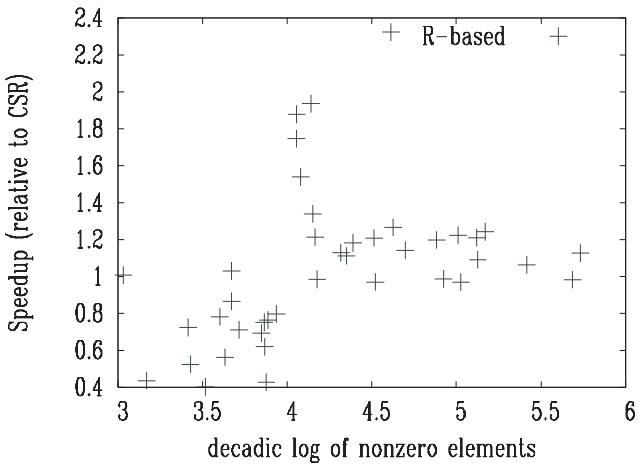

Fig. 14: The speedup estimated by an SW cache analyzer

Figs. 13, 14 and 15 show the speedups of $SpM{\times}V$ on real matrices.

- Slowdown of about 20 % was obtained only for small matrices that fit in the cache. This expected result follows from the fact that the R-based format for these matrices does not improve cache utilization and the matrix transformation destroys the naturally occurring locality.
- On the other hand, speedup of about 20 % was achieved for all large matrices (that do not fit in the cache).

This speedup is due to better cache utilization, for two main reasons:

- a lower number of compulsory misses due to compressed representation of the matrix,
- a lower number of trashing misses due to region traversing.

The big peaks in the plotted data occur for matrices "on the edge". These are large enough (do not fit in the cache) in the CSR format, but small enough (fit in the cache) in the R-based format.

3.2.5 Payoff iterations

The transformation into another format always carries some overhead. A useful question is whether the transformation into different formats pays off in real cases. To answer this

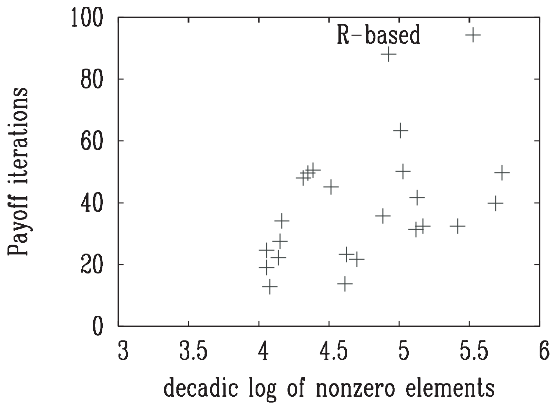

Fig. 16: Payoff iterations

question, we define the parameter *Payoff* as the ratio between the matrix transformation time and the difference between the time of *SpM×V* with the CSR format and the time of *SpM×V* with the R-based format. This denotes the number of executions of *SpM×V* to amortize the overhead of the matrix format transformation.

From Fig. 16, we can conclude that the *Payoff* is approximately 40 (iterations) for large matrices that do not fit into the cache.

3.2.6 Impact of the traversal on the performance

Another important question is whether and in what way the region traversal has a significant effect on the performance of SpM×V.

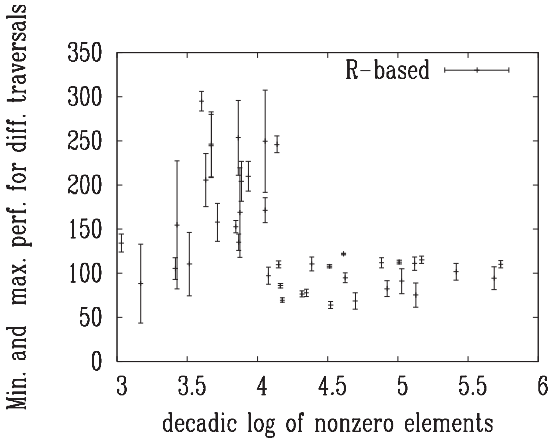

Fig. 17: The difference between the worst and best region traversal

Fig. 17 illustrates the importance of choosing a good traversal for various strategies in Fig. 4. The difference of the *SpM×V* performance was up to 100 % for small matrices and about 20 % for larger matrices. We can conclude that choosing the traversal has a significant effect on the performance.

4 Conclusions

In this paper, we have presented an unusual approach to accelerate sparse matrix-vector multiplication. This approach consists of 3 steps. First, we divide matrix *A* into nonempty disjoint square regions. Second, we choose a suitable storage format for each region. We introduce new sparse matrix R-based (R-XY or R-CSR) formats for additional acceleration of *SpM×V*. These new formats cause much fewer compulsory misses. Third, we present a methodology for finding a good region traversal. We compare the measured and predicted values. For all types of sufficiently large matrices arising from various technical disciplines, our approach gives a significant speedup. The matrix transformation is relatively fast and straightforward. Our measurements prove that matrix transformation pays off for a relatively small number *SpM×V*.

Acknowledgments

This research has been supported by the Czech Ministry of Education, Youth and Sport under research program MSM6840770014.

References

[1] Im, E.: *Optimizing the Performance of Sparse Matrix-Vector Multiplication – dissertation thesis*. Dissertation thesis, University of Carolina at Berkeley, 2001.

[2] Mellor-Crummey, J., Garvin, J.: Optimizing Sparse Matrix Vector Product Computations Using Unroll and Jam. *International Journal of High Performance Computing Applications*, Vol. **18** (2004), No. 2, p. 225–236.

[3] Vuduc, R., Demmel, J. W., Yelick, K. A., Kamil, S., Nishtala, R., Lee, B.: Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Proceedings of Supercomputing 2002*, Baltimore, MD, USA, November 2002.

[4] Tvrdík, P., Šimeček, I.: A new Diagonal Blocking Format and Model of Cache Behavior for Sparse Matrices. In *Parallel Processing and Applied Mathematics*, vol. **3911** of *Lecture Notes of Computer Science*, p. 164–171, Poznan, (Poland) 2006, Springer.

[5] White, J., Sadayappan, P.: On Improving the Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the 4th International Conference on High Performance Computing* (HiPC '97), p. 578–587. IEEE Computer Society, 1997.

[6] Tvrdík, P., Šimeček, I.: Software Cache Analyzer. In *Proceedings of CTU Workshop*, Vol. **9**, p. 180–181, Prague, Czech Republic, Mar. 2005.

Ing. Ivan Šimeček
phone: +420 224 357 268
e-mail: xsimecek@fel.cvut.cz

Department of Computer Science

Czech Technical University in Prague
Faculty of Electrical Engineering
Technická 2
166 27 Prague 6, Czech Republic